Available online at www.sciencedirect.com

SCIENCE DIRECT

Electronic Notes in Theoretical Computer Science 158 (2006) 99–121

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

The Linear Logical Abstract Machine

Eduardo Bonelli[1]

LIFIA, Fac. de Informática, UNLP, Argentina

Abstract

We derive an abstract machine from the Curry-Howard correspondence with a sequent calculus presentation of Intuitionistic Propositional Linear Logic. The states of the register based abstract machine comprise a low-level code block, a register bank and a dump holding suspended procedure activations. Transformation of natural deduction proofs into our sequent calculus yields a type-preserving compilation function from the Linear Lambda Calculus to the abstract machine. We prove correctness of the abstract machine with respect to the standard call-by-value evaluation semantics of the Linear Lambda Calculus.

Keywords: Linear Logic, Curry-Howard Isomorphism, Abstract Machine, Linear Lambda Calculus, Compilation

1 Introduction

Principally motivated by security concerns, logical and type theoretic foundations of low-level code (eg. typed assembly language [16] and bytecode [15]) and abstract machines (such as SECD-style machines [13] and the Java Virtual Machine [15]) have received considerable attention recently. Our interest is in the logical foundations of abstract machines: This paper presents a proof theoretic account for an abstract machine in the setting of Linear Logic [10] and establishes its correctness. The contributions of this work may be summed up as follows:

(i) We introduce a sequent calculus for intuitionistic propositional linear logic, the *linear sequential sequent calculus* (SS), and we show that the term assignment for SS is low-level code in which terms encoding lazy

[1] Email: eduardo@sol.info.unlp.edu.ar

doi:10.1016/j.entcs.2006.04.007

connectives introduce appropriate closures. SS is *sequential* [17,18] in the following sense: the succedent of each sequent which is the conclusion of an inference scheme is identical to the succedent of its major premise. For example, the inference schemes for tensor and with are:

$$\frac{\Gamma, A, B \vdash C}{\Gamma, A \otimes B \vdash C}\,(\otimes L) \qquad \frac{\Gamma, A \otimes B \vdash C}{\Gamma, A, B \vdash C}\,(\otimes R) \qquad \frac{\Gamma, A \vdash C}{\Gamma, A\&B \vdash C}\,(\&L1)$$

$$\frac{\Gamma, B \vdash C}{\Gamma, A\&B \vdash C}\,(\&L2) \qquad \frac{\Gamma, A\&B \vdash C \quad \Delta \vdash A \quad \Delta \vdash B}{\Gamma, \Delta \vdash C}\,(\&R)$$

Consequently, proofs in SS have a "principal" branch and inference schemes may be seen to operate on the antecedent of the sequents in this branch. This is reflected in the term assignment for these inference schemes (here x, y, z stand for registers):

$$\frac{\Gamma, y : A, z : B \vdash_c \mathcal{B} : C}{\Gamma, x : A \otimes B \vdash_c (y,z)\texttt{=unpair}(x)\texttt{;}\mathcal{B} : C}\,(\otimes L) \qquad \frac{\Gamma, z : A \otimes B \vdash_c \mathcal{B} : C}{\Gamma, x : A, y : B \vdash_c z\texttt{=pair}(x,y)\texttt{;}\mathcal{B} : C}\,(\otimes R)$$

$$\frac{\Gamma, x : A \vdash_c \mathcal{B} : C}{\Gamma, y : A\&B \vdash_c x\texttt{=fst}(y)\texttt{;}\mathcal{B} : C}\,(\&L1) \qquad \frac{\Gamma, x : B \vdash_c \mathcal{B} : C}{\Gamma, y : A\&B \vdash_c x\texttt{=snd}(y)\texttt{;}\mathcal{B} : C}\,(\&L2)$$

$$\frac{\Gamma, x : A\&B \vdash_c \mathcal{B} : C \quad \Delta \vdash_c \mathcal{B}_1 : A \quad \Delta \vdash_c \mathcal{B}_2 : B}{\vec{y} : \Delta, \Gamma \vdash_c x\texttt{=makeLPClos}(\mathcal{B}_1, \mathcal{B}_2, \vec{y})\texttt{;}\mathcal{B} : C}\,(\&R)$$

For example, $x\texttt{=makeLPClos}(\mathcal{B}_1, \mathcal{B}_2, \vec{y})\texttt{;}\mathcal{B}$ is a code block whose first instruction creates a closure associated to the lazy pair constructor "&" and deposits it in register x. The full set of instructions is described in Sec. 3. In formulating SS by following this pattern for the other connectives we note that all those whose computational interpretation is *eager* [2] such as the tensor, yield inference schemes with one hypothesis. However, all *lazy* connectives such as the of course modality "!" and with "&" require one or more additional hypothesis which represent additional code blocks with which appropriate closures are constructed.

(ii) From cut-elimination in SS we derive a register based abstract machine (the *Linear Logical Abstract Machine* or LLAM) whose states comprise a code block, a register bank and a dump holding suspended procedure activations. Each step in the cut-elimination proof corresponds to a reduction step in the LLAM. The LLAM includes instructions for creating functional, with and of course closures, executing closures, constructing and destructing eager pairs, duplicating and erasing registers, and returning from a call. Each machine state has a register bank, a code block and a dump, the latter of which encodes the application of the Induction Hypothesis in the proof of cut-elimination.

(iii) We show that the transformation of Natural Deduction proofs into SS yields a type-preserving *compilation function* from the *Linear Lambda Calculus* or λ^l (as presented by Wadler [24]) to our low-level code.

(iv) We prove the correctness of the LLAM with respect to evaluation in λ^l: If a term in λ^l evaluates to some canonical form, then its compilation reduces in the abstract machine to the corresponding compiled canonical form. The proof is rather standard in SECD-style [2] machines in which lambda terms themselves are executed (eg. [19]). In our case, however, in which low-level code different from lambda terms are executed, we need to introduce some additional machinery for the proof to go through, including an intermediate notion of evaluation (which we call lifted evaluation) where substitutions are managed explicitly.

Structure of the paper. We formulate SS in Section 2 and also address cut-elimination and the relation between SS and natural deduction. The syntax, type system and operational semantics of the LLAM is presented next. Section 4 briefly recalls λ^l and defines a compilation function from λ^l to the low-level code of the LLAM based on the proof transformation developed in the preceding section. This section ends with the proof of correctness of the LLAM. Finally, we conclude and suggest further research directions.

1.1 Related Work

The first abstract machines related to linear logic were introduced by Lafont [12] and Abramsky [2] (the "Linear SECD Machine" - LSECD). The Linear Abstract Machine (LAM) of Lafont is inspired by the categorical combinators of the theory of symmetric monoidal closed categories (modal types are encoded using the standard recursive encoding). Our LLAM is derived from cut-elimination in a sequent calculus. Moreover, in LAM there is no sharing and hence cells may be reclaimed immediately by the instructions of the machine. This is not favourable if sequential implementations are desired. The LLAM does allow sharing for exponential types. However, this must be achieved by introducing locations and tailored reduction schemes [2] which are not mirrored by our cut-elimination analysis. Regarding the LSECD Machine it is simply presented as a variant of Landin's SECD machine [13] without supplying further details on its foundations. Lincoln and Mitchell [14] sketch an abstract machine but do not define it precisely nor give any proofs for its properties. The work of Raffalli [20] and Ariola et al [3] is also relevant, how-

[2] In the light of the distinction proposed by Ager at al [1] one might call our machine a *virtual* machine since it has its own instruction set. This is in contrast to abstract machines which operate directly on λ-terms and which we refer to as "SECD-style" machines.

ever none of them address Linear Logic. Mackie investigates the use of the Linear Term Calculus [2] as the basis for a functional programming language by extending it with natural numbers, booleans, lists, iterators and recursion. The abstract machine he considers is an extension of the LSECD machine to cope with these new constructs. In a similar line of research, Wakeling and Runciman [25] also study functional programming based on a linear term calculus introduced by Wadler [23]. Danos et al [7] introduce two abstract machines in order to relate Hyland-Ong and Abramsky-Jagadeesan-Malacaria game semantics. The relation between these semantics is established by relating the machines: both implement linear head reduction in the lambda calculus. Alberti and Ritter [4] introduce a linear abstract machine with the aim of allowing sharing of non-linear types while ensuring the single-pointer property. However the logical foundations of the machine and its correctness is not studied. The approach to machine derivation by means of a Curry-Howard isomorphism on sequential sequent calculi introduced by Ohori [17,18] is the closest to this paper. However, it does not deal with Linear Logic and no proof of correctness for the abstract machine is provided. Further relevant work (although not dealing with abstract machines) is developed by Wadler et al [24,22] and Bierman et al [5] among others. An annotated bibliography on abstract machines is provided by Diehl et al [8].

2 Sequential Sequent Calculus

A proposition of ILL is either a propositional constant $\mathbf{N}$, a linear implication $A \multimap B$, a tensor product (or multiplicative conjunction) $A \otimes B$, a direct product (or additive conjunction) $A \& B$, or an exponential ("!" is called "bang" or "of course") $!A$. The standard computational interpretation [12,2] of these propositions as types may be informally described as follows. $\mathbf{N}$ is the type of the natural numbers. $A \multimap B$ is the type of functions which use their argument exactly once. $A \otimes B$ is the type of pairs in which, on its unique consumption, both components are used exactly once. $A \& B$ is the type of the pairs in which we must choose whether to use the first component or the second one, the other component is no longer available for selection once our choice is made. $!A$ is the type of values of type A that may be used as many times as we wish (possibly none).

A *context* Γ is a multiset of propositions and $\emptyset$ is the empty context. SS comprises four *logical judgements* (*sequents* for short):

$\Gamma \vdash_c A$	*Code block judgement*	$\vdash_v A$	*Value judgement*
$\vdash_e \Gamma$	*Environment judgement*	$\vdash A$	*Top-level judgement*

The axiom and inference schemes defining these judgements are given in Fig. 1. We write SS^c for the schemes defining the code block judgement (the reason for the qualifier "code block" is explained in Sec. 3). A proof of a code block judgement is called a *code block proof* (and likewise for the remaining judgements). The *major premise* of an inference scheme is the leftmost premise, the others (if present) are the *minor premises*. In a sequent $\Gamma \vdash_c A$, we call Γ the *antecedent* and A the *succedent* of the sequent. A sample proof in SS^c (of $!(A\&B) \vdash !A \otimes !B$) is

$$\dfrac{\dfrac{\dfrac{}{!A \otimes !B \vdash_c !A \otimes !B}\,(\otimes L)}{!A, !B \vdash_c !A \otimes !B}\,(!R) \qquad \dfrac{\dfrac{\dfrac{A \vdash_c A}{A\&B \vdash_c A}\,(\&L1)}{!(A\&B) \vdash_c A}\,(!L) \qquad \dfrac{\dfrac{B \vdash_c B}{A\&B \vdash_c B}\,(\&L2)}{!(A\&B) \vdash_c B}\,(!L)}{!B, !(A\&B) \vdash_c !A \otimes !B}\,(!R)}{!(A\&B), !(A\&B) \vdash_c !A \otimes !B}\,(\otimes L)}{!(A\&B) \vdash_c !A \otimes !B}\,(C)$$

We now briefly describe these judgements. First we point out that SS^c is *sequential* in the sense discussed in the introduction. The *major premise path* of a proof is the path of sequents obtained by traversing the major premise of each inference step in the proof, from the end sequent to the initial sequent. Second, SS^c is equivalent to the standard sequent calculus presentation of ILL as may be verified by a straightforward induction on the proofs of the judgements in question.

Lemma 2.1 *$\Gamma \vdash_c A$ is provable in SS^c iff $\Gamma \vdash A$ is provable in the standard sequent calculus presentation of ILL.*

Regarding the axioms and inference schemes for the value judgement there is one inference scheme per connective. Note that lazy type constructors, namely with and of course, have an environment judgement and a code block judgement as hypothesis. These are used for constructing the appropriate closures. An environment is either an empty environment (written $\emptyset$) or a multiset of propositions each of which is provable using the value judgements.

In SS only a top-level cut rule, called *multicut* [3] (cf. Fig. 1), is available. Multicut is applied at the top-level and simultaneously cuts all the formulas in the antecedent of $\Gamma \vdash_c A$. Furthermore, the sequential nature of SS^c allows the cut-elimination process to always replace a multicut with another multicut. This induces two desired properties of the abstract machine resulting from cut-elimination: reduction is on closed machine states and always takes place at the root. Regarding cut-elimination we have:

[3] In the sequel we shall speak of "cut-elimination" instead of "multicut-elimination".

Inference Schemes for Code Blocks

$$\frac{}{A \vdash_c A}\,(axiom) \qquad \frac{\Gamma, \mathbf{N} \vdash_c A}{\Gamma \vdash_c A}\,(nat)$$

$$\frac{\Gamma, B \vdash_c C}{\Gamma, A, A \multimap B \vdash_c C}\,(\multimap L) \qquad \frac{\Gamma, A \multimap B \vdash_c C \quad \Delta, A \vdash_c B}{\Gamma, \Delta \vdash_c C}\,(\multimap R)$$

$$\frac{\Gamma, A, B \vdash_c C}{\Gamma, A \otimes B \vdash_c C}\,(\otimes L) \qquad \frac{\Gamma, A \otimes B \vdash_c C}{\Gamma, A, B \vdash_c C}\,(\otimes R) \qquad \frac{\Gamma, A \vdash_c C}{\Gamma, A\&B \vdash_c C}\,(\&L1) \qquad \frac{\Gamma, B \vdash_c C}{\Gamma, A\&B \vdash_c C}\,(\&L2)$$

$$\frac{\Gamma, A\&B \vdash_c C \quad \Delta \vdash_c A \quad \Delta \vdash_c B}{\Gamma, \Delta \vdash_c C}\,(\&R) \qquad \frac{\Gamma, !B \vdash_c C \quad !\Delta \vdash_c B}{\Gamma, !\Delta \vdash_c C}\,(!R)$$

$$\frac{\Gamma, A \vdash_c C}{\Gamma, !A \vdash_c C}\,(!L) \qquad \frac{\Gamma, !A, !A \vdash_c C}{\Gamma, !A \vdash_c C}\,(C) \qquad \frac{\Gamma \vdash_c C}{\Gamma, !A \vdash_c C}\,(W)$$

Inference Schemes for Value Judgements

$$\frac{}{\vdash_v \mathbf{N}}\,(natV) \qquad \frac{\vdash_v A \quad \vdash_v B}{\vdash_v A \otimes B}\,(\otimes V) \qquad \frac{\vdash_e \Delta \quad \Delta, A \vdash_c B}{\vdash_v A \multimap B}\,(\multimap V)$$

$$\frac{\vdash_e \Delta \quad \Delta \vdash_c A \quad \Delta \vdash_c B}{\vdash_v A\&B}\,(\&V) \qquad \frac{\vdash_e !\Delta \quad !\Delta \vdash_c B}{\vdash_v !B}\,(!V)$$

Inference Schemes for Env. Judgements **Top-Level Judgement**

$$\frac{}{\vdash_e \emptyset}\,(nilE) \qquad \frac{\vdash_e \Gamma \quad \vdash_v A}{\vdash_e \Gamma, A}\,(consE) \qquad \frac{\vdash_e \Gamma \quad \Gamma \vdash_c A}{\vdash A}\,(mcut)$$

Fig. 1. Schemes for values and environments, and multicut

Proposition 2.2 (Cut-elimination) *Cut-elimination is strongly normalizing (SN) and confluent.*

Confluence is trivial since the cut-elimination process introduces at most one application of multicut at each step. SN, which we address below, can be proved using standard reducibility arguments [9,18]. Define (recursively) a family of *reducible code block (value, environment) proofs* indexed by a sequent (proposition, context, resp): $Red_c(\Gamma \vdash_c A)$ is the set of reducible code block proofs with endsequent $\Gamma \vdash_c A$, $Red_v(A)$ is the set of reducible value proofs of $\vdash_v A$ and $Red_e(\Gamma)$ is the set of reducible environment proofs of $\vdash_e \Gamma$. Intuitively, a value proof $\mathcal{V}$ is reducible if all the code block and environment

proofs of $\mathcal{V}$ are reducible; in turn, an environment proof is reducible if each value subproof in it is reducible; finally a code block proof $\mathcal{C}$ is reducible if all multicuts

$$\frac{\mathcal{E} \quad \mathcal{C}}{\vdash A}\,(mcut) \tag{1}$$

with reducible environment proofs $\mathcal{E}$ can be transformed to a reducible value proof of A.

Definition 2.3 (Families of reducible proofs) *The family of sets of reducible proofs $Red_v(A)$, $Red_e(\Gamma)$ and $Red_c(\Gamma \vdash_c A)$ is defined recursively:*

- *Value proofs*
  - $\dfrac{}{\vdash_v \mathbf{N}}\,(natV) \in Red_v(\mathbf{N})$
  - $\dfrac{\mathcal{V}_1(\vdash_v A) \quad \mathcal{V}_2(\vdash_v B)}{\vdash_v A \otimes B}\,(\otimes V) \in Red_v(A \otimes B)$ *if* $\mathcal{V}_1 \in Red_v(A)$ *and* $\mathcal{V}_2 \in Red_v(B)$
  - $\dfrac{\mathcal{E}(\vdash_e \Delta) \quad \mathcal{C}(\Delta, A \vdash_c B)}{\vdash_v A \multimap B}\,(\multimap V) \in Red_v(A \multimap B)$ *if* $\mathcal{E} \in Red_e(\Delta)$ *and* $\mathcal{C} \in Red_c(\Delta, A \vdash_c B)$
  - $\dfrac{\mathcal{E}(\vdash_e \Delta) \quad \mathcal{C}_1(\Delta \vdash_c A) \quad \mathcal{C}_2(\Delta \vdash_c B)}{\vdash_v A\&B}\,(\&V) \in Red_v(A\&B)$ *if* $\mathcal{E} \in Red_e(\Delta)$, $\mathcal{C}_1 \in Red_c(\Delta \vdash_c A)$ *and* $\mathcal{C}_2 \in Red_c(\Delta \vdash_c B)$
  - $\dfrac{\mathcal{E}(\vdash_e !\Delta) \quad \mathcal{C}(!\Delta \vdash_c B)}{\vdash_v !B}\,(!V) \in Red_v(!B)$ *if* $\mathcal{E} \in Red_e(!\Delta)$ *and* $\mathcal{C} \in Red_c(!\Delta \vdash_c B)$
- *Environment proofs*
  - $\dfrac{}{\vdash_e \emptyset}\,(nilE) \in Red_e(\emptyset)$
  - $\dfrac{\mathcal{E}(\vdash_e \Gamma) \quad \mathcal{V}(\vdash_v A)}{\vdash_e \Gamma, A}\,(consE) \in Red_e(\Gamma, A)$ *if* $\mathcal{E} \in Red_e(\Gamma)$ *and* $\mathcal{V} \in Red_v(A)$
- *Code block proofs*
  - $\mathcal{C}(\Gamma \vdash_c A) \in Red_c(\Gamma \vdash_c A)$ *if for every* $\mathcal{E} \in Red_e(\Gamma)$, *the top-level proof (1) is transformed to a value proof in* $Red_v(A)$.

We now consider the proof of SN of Cut-Elimination by showing that if $\mathcal{C}(\Gamma \vdash_c A)$ is a code block proof, then $\mathcal{C} \in Red_c(\Gamma \vdash_c A)$. It makes use of the fact that if $\mathcal{E} \in Red_e(\Gamma, \Delta)$, then there exist environment proofs $\mathcal{E}_1$ and $\mathcal{E}_2$ such that $\mathcal{E}_1 \in Red_e(\Gamma)$ and $\mathcal{E}_2 \in Red_e(\Delta)$.

Proof. By induction on the proof of $\mathcal{C}(\Gamma \vdash_c A)$. Case analysis is performed on the last inference scheme used to prove $\mathcal{C}$ in (1). We give two sample cases.

- A right introduction rules introduces a new value proof which is used to extend the environment proof $\mathcal{E}$ of (1). A new multicut is then introduced using the extended environment proof and the subproof just above the conclusion of $\mathcal{C}$. As an example, here is the case of $(\&R)$. The proof ends

in:

$$\dfrac{\mathcal{E}_1(\vdash_e \Gamma, \Delta) \qquad \dfrac{\mathcal{C}_3(\Gamma, A\&B \vdash_c C) \quad \mathcal{C}_1(\Delta \vdash_c A) \quad \mathcal{C}_2(\Delta \vdash_c B)}{\Gamma, \Delta \vdash_c C}\,(\&R)}{\vdash C}\,(mcut)$$

This may be transformed into

$$\dfrac{\dfrac{\mathcal{E}_1(\vdash_e \Gamma) \qquad \dfrac{\mathcal{E}_2(\vdash_e \Delta) \quad \mathcal{C}_1(\Delta \vdash_c A) \quad \mathcal{C}_2(\Delta \vdash_c B)}{\vdash_v A\&B}\,(\&V)}{\vdash_e \Gamma, A\&B}\,(consE) \qquad \Gamma, A\&B \vdash_c C}{\vdash C}\,(mcut)$$

to whom we may apply the IH and conclude.

- A left introduction rule resorts to an existing value proof in $\mathcal{E}$, a subproof of $\mathcal{C}$ and a new multicut to compute a new value proof (representing an intermediate result). While this intermediate result is computed the elimination of the original multicut is suspended (this is to be encoded in the dump of our upcoming abstract machine). Once the new value proof is obtained, a new environment proof is constructed using it, and a new multicut with a subproof of $\mathcal{C}$ is introduced. We illustrate this with the case of $(\&L1)$. Suppose the proof ends in

$$\dfrac{\mathcal{E}_1(\vdash_e \Gamma, A\&B) \qquad \dfrac{\mathcal{C}_1(\Gamma, A \vdash_c C)}{\Gamma, A\&B \vdash_c C}\,(\&L1)}{\vdash C}\,(mcut)$$

From $\mathcal{E}_1(\vdash_e \Gamma, A\&B)$ we know that there exists $\mathcal{E}_2$ such that $\mathcal{E}_2(\vdash_e \Gamma) \in Red_e(\Gamma)$ and that there must be a value proof of $\vdash_v A\&B$. Moreover, this proof must have the form

$$\dfrac{\mathcal{E}_2(\vdash_e \Delta) \quad \mathcal{C}_1(\Delta \vdash_c A) \quad \mathcal{C}_2(\Delta \vdash_c B)}{\vdash_v A\&B}\,(\&V)$$

for some Δ and $\mathcal{E}_2 \in Red_e(\Delta)$. Therefore the following top-level proof

$$\dfrac{\mathcal{E}_1(\vdash_e \Delta) \quad \Delta \vdash_c A}{\vdash A}\,(mcut)$$

may be transformed into a proof $\mathcal{V}_1 \in Red_v(A)$. This computes our intermediate result. Using the resulting value proof, the following environment proof $\mathcal{E}_3$

$$\dfrac{\mathcal{E}_2(\vdash_e \Gamma) \quad \mathcal{V}_1(\vdash_v A)}{\vdash_e \Gamma, A}\,(consE)$$

is seen to be in $Red_e(\Gamma, A)$. As a consequence we may apply the IH to the proof

$$\dfrac{\mathcal{E}_1(\vdash_e \Gamma, A) \quad \mathcal{C}_1(\Gamma, A \vdash_c C)}{\vdash C}\,(mcut)$$

$$\frac{}{A \vdash A}\,(axiom) \qquad \frac{}{\vdash \mathbf{N}}\,(nat) \qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \multimap B}\,(\multimap I) \qquad \frac{\Gamma \vdash A \multimap B \quad \Gamma' \vdash A}{\Gamma, \Gamma' \vdash B}\,(\multimap E)$$

$$\frac{\Gamma \vdash A \quad \Gamma' \vdash B}{\Gamma, \Gamma' \vdash A \otimes B}\,(\otimes I) \qquad \frac{\Gamma \vdash A \otimes B \quad \Gamma', A, B \vdash C}{\Gamma, \Gamma' \vdash C}\,(\otimes E)$$

$$\frac{\Gamma \vdash A \& B}{\Gamma \vdash A}\,(\&E1) \qquad \frac{\Gamma \vdash A \& B}{\Gamma \vdash B}\,(\&E2) \qquad \frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \& B}\,(\&I)$$

$$\frac{!\Gamma \vdash A}{!\Gamma \vdash !A}\,(!I) \qquad \frac{\Gamma \vdash !B \quad \Gamma', B \vdash A}{\Gamma, \Gamma' \vdash A}\,(!E)$$

$$\frac{\Gamma \vdash !A \quad \Gamma', !A, !A \vdash C}{\Gamma, \Gamma' \vdash C}\,(C) \qquad \frac{\Gamma \vdash !A \quad \Gamma' \vdash C}{\Gamma, \Gamma' \vdash C}\,(W)$$

Fig. 2. Natural Deduction for ILL

and conclude.

We now address the relation between SS and Natural Deduction for ILL (ND, cf. Fig.2). The standard approach transforming ND proofs into Sequent Calculus proofs is to map inference schemes in ND that introduce a connective in the former system to right introduction schemes in the latter, and those that eliminate a connective to left introduction schemes plus possible applications of (the standard rule) *cut*. We follow the same approach except that since standard cut-elimination in SS^c is straightforward due to its sequential nature[4], all cuts are eliminated directly by means of a *proof transformer*.

A *proof transformer* $\mathcal{P}$ from Γ to Δ in SS, written $\mathcal{P}[] : \Delta \Rightarrow \Gamma$, is a code block proof with a hole $\Box_\Gamma$ subscripted with a context at the initial sequent on the major premise path and whose end sequent has Δ as context. We write $\mathcal{P}[\mathcal{C}]$ for the proof obtained by filling the hole in $\mathcal{P}$ with the code block proof $\mathcal{C}$. We view $\mathcal{P}[] : \Delta \Rightarrow \Gamma$ as a proof transformer in the sense that it takes a proof $\mathcal{C}(\Gamma \vdash_c A)$ and transforms it to a proof of $\Delta \vdash_c A$. The following result relates provability in ND and existence of proof transformers (its proof proceeds by induction on proof of $\Gamma \vdash A$).

Lemma 2.4 *If $\Gamma \vdash A$ is provable in ND, then there exists a proof transformer $\mathcal{P}[] : \Gamma', \Gamma \Rightarrow \Gamma', A$, for all Γ'.*

Proposition 2.5 *If $\Gamma \vdash A$ is provable in ND, then $\Gamma \vdash A$ is provable in SS^c.*

[4] The cut formula may be seen never to change: occurrences of the cut inference scheme are pushed downwards towards the root of the proof and then eliminated once they reach the axioms.

Proof. If $\Gamma \vdash A$ is provable in ND, then by Lemma 2.4 there exists a proof transformer $\mathcal{P}[] : \Gamma \Rightarrow A$. Hence $\mathcal{P}[A \vdash A](\Gamma \vdash A)$ is a proof of $\Gamma \vdash A$ in SS^c.

3 The Linear Logical Abstract Machine

3.1 The Code Language and Type System

A *code block* is a sequence of instructions. We let $\mathcal{B}$ range over code blocks, ι over instructions and x, y, z over a countable infinite set $\mathcal{R}$ of registers. The grammar defining code blocks and instructions is:

$$\begin{aligned}
\mathcal{B} &::= \iota;\mathcal{B} \mid \texttt{Return}(x) \\
\iota &::= x{=}\underline{n} \mid x{=}\texttt{makeClos}(\mathcal{B}, y, \vec{z}) \mid x{=}\texttt{call}\, y\, \texttt{with}\, z \\
&\quad \mid x{=}\texttt{pair}(y,z) \mid (x,y){=}\texttt{unpair}(z) \\
&\quad \mid x{=}\texttt{makeLPClos}(\mathcal{B},\mathcal{B},\vec{y}) \mid x{=}\texttt{fst}(y) \mid x{=}\texttt{snd}(y) \\
&\quad \mid x{=}\texttt{makeOCClos}(\mathcal{B},\vec{y}) \mid x{=}\texttt{read}(y) \mid (x,y){=}\texttt{copy}\, z \mid \texttt{kill}(x)
\end{aligned}$$

The code language of the LLAM is a register transfer language with an unbounded number of registers. A typical instruction has the form $x{=}\texttt{op}(\vec{y})$ where op is the operation code, x is the destination register and $\vec{y}$ are the argument registers. A brief description of (some of) the instructions follows.

$\texttt{Return}(x)$ returns to the caller with the contents of register x. $x{=}\underline{n}$ assigns the numeral $\underline{n}$ to register x. $(x,y){=}\texttt{unpair}(z)$ destructs the eager pair residing in register z: the first component is placed in register x while the second component is placed in register y. Note that, in accordance with the eager nature of the $\otimes$ connective in ILL, it is not possible to project only the first (or only the second) component of an eager pair. $x{=}\texttt{makeLPClos}(\mathcal{B}_1,\mathcal{B}_2,\vec{y})$ creates a lazy pair. Such pairs are represented as *lazy pair closures* which consist of a pair of code blocks $\mathcal{B}_1$ and $\mathcal{B}_2$ together with the current register bank restricted to the registers in $\vec{y}$. This closure is placed in register x and registers $\vec{y}$ are no longer available since they were consumed in order to construct the closure. The remaining instructions may be understood along similar lines.

The code language is typed. A *typing judgement for code blocks* is an expression of the form $\Gamma \vdash_c \mathcal{B} : A$, where Γ (the *typing context*) is a multiset of expressions of the form $x_i : A_i$, $1 \leq i \leq n$, where x_i is a register, A_i is a proposition in ILL and the x_i are all distinct, $\mathcal{B}$ is a code block and A is a proposition in ILL. The typing schemes (samples of which were given in the introduction) are the logical schemes that define the code block judgement described in Sec. 2 decorated with typing information (cf. Fig. 3). We sometimes write $\vec{x} : \Gamma$ when $\Gamma = \{x_1 : A_1, \ldots, x_n : A_n\}$ and $\vec{x} = x_1, \ldots, x_n$. Also, $\Gamma, x : A$ is shorthand for $\Gamma \cup \{x : A\}$ assuming $x \notin \mathsf{Dom}(\Gamma)$.

Typing Schemes for Code Blocks

$$\frac{}{x : A \vdash_c \texttt{Return}(x) : A}\ (axiom) \qquad \frac{\Gamma, x : \mathbf{N} \vdash_c \mathcal{B} : C}{\Gamma \vdash_c x{=}\underline{n}\,;\mathcal{B} : C}\ (nat)$$

$$\frac{\Gamma, z : B \vdash_c \mathcal{B} : C}{\Gamma, y : A, x : A \multimap B \vdash_c z{=}\texttt{call}\,x\,\texttt{with}\,y\,;\mathcal{B} : C}\ (\multimap L) \qquad \frac{\Gamma, x : A \multimap B \vdash_c \mathcal{B} : C \quad \Delta, z : A \vdash_c C : B}{\Gamma, \vec{y} : \Delta \vdash_c x{=}\texttt{makeClos}(C, z, \vec{y})\,;\mathcal{B} : C}\ (\multimap R)$$

$$\frac{\Gamma, y : A, z : B \vdash_c \mathcal{B} : C}{\Gamma, x : A \otimes B \vdash_c (y,z){=}\texttt{unpair}(x)\,;\mathcal{B} : C}\ (\otimes L) \qquad \frac{\Gamma, z : A \otimes B \vdash_c \mathcal{B} : C}{\Gamma, x : A, y : B \vdash_c z{=}\texttt{pair}(x,y)\,;\mathcal{B} : C}\ (\otimes R)$$

$$\frac{\Gamma, x : A \vdash_c \mathcal{B} : C}{\Gamma, y : A\&B \vdash_c x{=}\texttt{fst}(y)\,;\mathcal{B} : C}\ (\&L1) \qquad \frac{\Gamma, x : B \vdash_c \mathcal{B} : C}{\Gamma, y : A\&B \vdash_c x{=}\texttt{snd}(y)\,;\mathcal{B} : C}\ (\&L2)$$

$$\frac{\Gamma, A\&B \vdash_c \mathcal{B} : C \quad \Delta \vdash_c \mathcal{B}_1 : A \quad \Delta \vdash_c \mathcal{B}_2 : B}{\vec{y} : \Delta, \Gamma \vdash_c x{=}\texttt{makeLPClos}(\mathcal{B}_1, \mathcal{B}_2, \vec{y})\,;\mathcal{B} : C}\ (\&R)$$

$$\frac{\Gamma, y : A \vdash_c \mathcal{B} : C}{\Gamma, x : !A \vdash_c y{=}\texttt{read}(x)\,;\mathcal{B} : C}\ (!L) \qquad \frac{\Gamma, x : !B \vdash_c \mathcal{B} : C \quad !\Delta \vdash_c C : B}{\vec{y} : !\Delta, \Gamma \vdash_c x{=}\texttt{makeOCClos}(C, \vec{y})\,;\mathcal{B} : C}\ (!R)$$

$$\frac{\Gamma, x : !A, y : !A \vdash_c \mathcal{B} : C}{\Gamma, z : !A \vdash_c (x,y){=}\texttt{copy}\,z\,;\mathcal{B} : C}\ (C) \qquad \frac{\Gamma \vdash_c \mathcal{B} : C}{\Gamma, x : !A \vdash_c \texttt{kill}(x)\,;\mathcal{B} : C}\ (W)$$

Typing Schemes for Values, Dumps and Machine States

$$\frac{}{\vdash_v \underline{n} : \mathbf{N}}\ (natV) \qquad \frac{\vdash_v v_1 : A \quad \vdash_v v_2 : B}{\vdash_v (v_1, v_2) : A \otimes B}\ (\otimes V) \qquad \frac{\vdash_e R : \Delta \quad \Delta, x : A \vdash_c \mathcal{B} : B}{\vdash_v Clos(B, x, R) : A \multimap B}\ (\multimap V)$$

$$\frac{\vdash_e R : \Delta \quad \Delta \vdash_c \mathcal{B}_1 : A \quad \Delta \vdash_c \mathcal{B}_2 : B}{\vdash_v LPClos(\mathcal{B}_1, \mathcal{B}_2, R) : A\&B}\ (\&V) \qquad \frac{\vdash_e R : !\Delta \quad !\Delta \vdash_c \mathcal{B} : B}{\vdash_v OCClos(\mathcal{B}, R) : !B}\ (!V)$$

$$\frac{}{\vdash_e \emptyset : \emptyset}\ (nilE) \qquad \frac{\vdash_e R : \Gamma \quad \vdash_v v : A}{\vdash_e R[x := v] : \Gamma, x : A}\ (consE)$$

$$\frac{}{\vdash_d \emptyset : \emptyset}\ (nilD) \qquad \frac{\vdash_e R : \Gamma \quad \Gamma, x : A \vdash_v \mathcal{B} : C \quad \vdash_d D}{\vdash_d [\mathcal{B}, R, x] \cdot D}\ (consD) \qquad \frac{\vdash_e R : \Delta \quad \Delta \vdash_c \mathcal{B} : C \quad \vdash_d D}{\vdash_{ms} \langle \mathcal{B}, R, D \rangle}\ (MS)$$

Fig. 3. Typing schemes for LLAM code blocks, values, dumps and machine states

3.2 Machine Architecture

A *register bank* $R, R', \ldots$ is a mapping with finite domain that associates *values* to registers. A value is one of the following: a *numeral* $\underline{n}$, an *eager pair* of values $\langle v_1, v_2 \rangle$, a *function closure* $Clos(\mathcal{B}, x, R)$, an *of course closure* $OCClos(\mathcal{B}, R)$ or a *lazy pair closure* $LPClos(\mathcal{B}_1, \mathcal{B}_2, R)$. Just like code blocks, values are also typed (bottom of Fig. 3). A *typing judgement for values* is an expression of the form $\vdash_v v : A$ where v is a value and A is a proposition in ILL.

A *dump* is a stack of suspended procedure activations: $[\mathcal{B}_1, R_1, x_1] \cdot \ldots \cdot [\mathcal{B}_n, R_n, x_n]$. Each procedure activation consists of a code block, a register bank and a register that shall hold the return value of the computation that caused suspension of execution. A *machine state* is a triple $\langle \mathcal{B}, R, D \rangle$ where $\mathcal{B}$ is a code block, R is a register bank and D is a dump. In LLAM machine states are *typed* (Fig. 3). A machine state $\langle \mathcal{B}, R, D \rangle$ is well-typed if the dump D is well-typed and there exists a typing context Δ and a type C such that the register bank is well-typed with type Δ and the code block is well-typed under typing context Δ with type C. The typing schemes for environments are self explanatory. Regarding those for dumps, each suspended procedure activation should be well-typed. If $\mathsf{Dom}(R)$ denotes the domain of R, then we write $R[x := v]$ for the register bank $R \cup \{x = v\}$, if $x \notin \mathsf{Dom}(R)$, otherwise $R[x := v]$ is undefined.

Before defining the operational semantics of the LLAM we need some additional notions. $R(x)$ denotes the value assigned to register x by R assuming $x \in \mathsf{Dom}(R)$. The *restriction of* R *to* $\vec{y}$, written $R \mid_{\vec{y}}$, is defined as R' if $\vec{y} \in \mathsf{Dom}(R)$, where $R'(x) = R(x)$ if $x \in \vec{y}$; $R'(x)$ is undefined otherwise. The *deletion of* $\vec{y}$ *from* R, written $R \backslash_{\vec{y}}$, is defined as $R - \{y_1 := R(y_1), \ldots, y_n := R(y_n)\}$ if $\vec{y} = y_1, \ldots, y_n \in \mathsf{Dom}(R)$; otherwise it is undefined.

The operational semantics of the LLAM is defined as a binary relation on states: $\langle \mathcal{B}, R, D \rangle$ *reduces to* $\langle \mathcal{B}', R', D' \rangle$ if $\langle \mathcal{B}, R, D \rangle \rightarrow \langle \mathcal{B}', R', D' \rangle$ according to the *reduction schemes* in Fig. 4. An *initial state* is of the form $\langle \mathcal{B}, R, \emptyset \rangle$ and a *final state* is one of the form $\langle \epsilon, \{x = v\}, \emptyset \rangle$, where ϵ denotes the empty sequence of instructions. This relation is obtained from the cut-elimination (Prop. 2.2) and as a consequence the following result holds immediately.

Proposition 3.1 (Type Safety) *If* $\vdash_{ms} \langle \mathcal{B}, R, D \rangle$ *and* $\langle \mathcal{B}, R, D \rangle$ *reduces to* $\langle \mathcal{B}', R', D' \rangle$, *then* $\vdash_{ms} \langle \mathcal{B}', R', D' \rangle$.

Furthermore, a typed machine state that is not a final state can always progress towards one. Its proof follows from a simple case analysis on the last typing scheme in the typing proof of $\mathcal{B}$.

Proposition 3.2 (Progress) *If* $\vdash_{ms} \langle \mathcal{B}, R, D \rangle$ *and* $\langle \mathcal{B}, R, D \rangle$ *is not a final*

$$\langle \texttt{Return}(x), \{x = v\}, \emptyset \rangle \rightarrow \langle \epsilon, \{x = v\}, \emptyset \rangle$$
$$\langle x\texttt{=}n\texttt{;}\mathcal{B}, R, D \rangle \rightarrow \langle \mathcal{B}, R[x := n], D \rangle$$
$$\langle x\texttt{=makeClos}(\mathcal{C}, w, \vec{y})\texttt{;}\mathcal{B}, R, D \rangle \rightarrow \langle \mathcal{B}, R \setminus_{\vec{y}} [x := Clos(\mathcal{C}, w, R\,|_{\vec{y}})], D \rangle$$
$$\langle z\texttt{=call}\, x\, \texttt{with}\, y\texttt{;}\mathcal{B}, R[x := Clos(\mathcal{C}, w, R')], D \rangle \rightarrow \langle \mathcal{C}, R'[w := R(y)], [\mathcal{B}, R\setminus_{x,y}, z] \cdot D \rangle$$
$$\langle \texttt{Return}(x), \{x = v\}, [\mathcal{B}, R, z] \cdot D \rangle \rightarrow \langle \mathcal{B}, R[z := v], D \rangle$$

$$\langle x\texttt{=pair}(y,z)\texttt{;}\mathcal{B}, R, D \rangle \rightarrow \langle \mathcal{B}, R \setminus_{y,z} [x := \langle R(y), R(z)\rangle], D \rangle$$
$$\langle (y,z)\texttt{=unpair}(x)\texttt{;}\mathcal{B}, R[x := \langle v_1, v_2\rangle], D \rangle \rightarrow \langle \mathcal{B}, R \setminus_{x} [y := v_1][z := v_2], D \rangle$$

$$\langle x\texttt{=makeLPClos}(\mathcal{B}_1, \mathcal{B}_2, \vec{y})\texttt{;}\mathcal{B}, R, D \rangle \rightarrow \langle \mathcal{B}, R \setminus_{\vec{y}} [x := LPClos(\mathcal{B}_1, \mathcal{B}_2, R\,|_{\vec{y}})], D \rangle$$
$$\langle x\texttt{=fst}(y)\texttt{;}\mathcal{B}, R[y := LPClos(\mathcal{B}_1, \mathcal{B}_2, R')], D \rangle \rightarrow \langle \mathcal{B}_1, R', [\mathcal{B}, R\setminus_{y}, x] \cdot D \rangle$$
$$\langle x\texttt{=snd}(y)\texttt{;}\mathcal{B}, R[y := LPClos(\mathcal{B}_1, \mathcal{B}_2, R')], D \rangle \rightarrow \langle \mathcal{B}_2, R', [\mathcal{B}, R\setminus_{y}, x] \cdot D \rangle$$

$$\langle x\texttt{=makeOCClos}(\mathcal{C}, \vec{y})\texttt{;}\mathcal{B}, R, D \rangle \rightarrow \langle \mathcal{B}, R \setminus_{\vec{y}} [x := OCClos(\mathcal{C}, R\,|_{\vec{y}})], D \rangle$$
$$\langle x\texttt{=read}(y)\texttt{;}\mathcal{B}, R[x := OCClos(\mathcal{C}, R')], D \rangle \rightarrow \langle \mathcal{C}, R', [\mathcal{B}, R\setminus_{y}, x] \cdot D \rangle$$
$$\langle (x,y)\texttt{=copy}\, z\texttt{;}\mathcal{B}, R, D \rangle \rightarrow \langle \mathcal{B}, R \setminus_{z} [x := R(z)][y := R(z)], D \rangle$$
$$\langle \texttt{kill}(x)\texttt{;}\mathcal{B}, R, D \rangle \rightarrow \langle \mathcal{B}, R\setminus_{x}, D \rangle$$

Fig. 4. Operational semantics of the LLAM

state, then there exists $\langle \mathcal{B}', R', D' \rangle$ *such that* $\langle \mathcal{B}, R, D \rangle \rightarrow \langle \mathcal{B}', R', D' \rangle$.

4 Compilation of λ^l and Correctness of the LLAM

This section introduces λ^l and a function that compiles a term in λ^l into code for the LLAM and then proves correctness of the LLAM. Since the syntax of terms in λ^l is not context free, we introduce an auxiliary syntactic category of *preterms* [2]. If U, V, W range over finite sets of variables, then we write $\mathcal{T}_U$ for the set of preterms with variables in U. The types of λ^l are the propositions of ILL. In particular, $\mathbf{N}$ is a base type. Γ is a *typing context*. The (λ^l-)*terms* are those preterms $M \in \mathcal{T}_U$ such that there exists Γ ($\mathsf{Dom}(\Gamma) = U$) and A such that $\Gamma \vdash M : A$ is provable using the standard axiom and inference schemes obtained from decorating the ND schemes (cf. Fig. 5).

As has been noted elsewhere [20,17,18], the transformation of natural deduction proofs to sequent calculus proofs yields a compilation function. Indeed, the proof of Prop. 2.5 yields the compilation function from λ^l-terms to *partial code blocks* (code blocks without the last `Return` instruction) shown in Fig. 6. Given a λ^l-term $M \in \mathcal{T}_U$ such that $\Gamma \vdash M : A$ is provable, its compilation is denoted $\textsc{Comp}(U \mid M \mid z)$. This shall return a partial code block which when suffixed with the instruction $\texttt{Return}(z)$ yields a proper code block that returns the compiled value of M in z. We use the symbol ϵ to denote

$$\frac{}{x : A \vdash x : A}\,(axiom) \qquad \frac{}{\vdash \underline{n} : \mathbf{N}}\,(nat)$$

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x : A.M : A \multimap B}\,(\multimap I) \qquad \frac{\Gamma \vdash M : A \multimap B \quad \Gamma' \vdash N : A}{\Gamma, \Gamma' \vdash M\,N : B}\,(\multimap E)$$

$$\frac{\Gamma \vdash M : A \quad \Gamma' \vdash N : B}{\Gamma, \Gamma' \vdash M \otimes N : A \otimes B}\,(\otimes I) \qquad \frac{\Gamma \vdash M : A \otimes B \quad \Gamma', x : A, y : B \vdash N : C}{\Gamma, \Gamma' \vdash let\ M\ be\ x \otimes y\ in\ N : C}\,(\otimes E)$$

$$\frac{\Gamma \vdash M : A \& B}{\Gamma \vdash fst(M) : A}\,(\&E1) \qquad \frac{\Gamma \vdash M : A \& B}{\Gamma \vdash snd(M) : B}\,(\&E2) \qquad \frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B}{\Gamma \vdash M \& N : A \& B}\,(\&I)$$

$$\frac{!\Gamma \vdash M : A}{!\Gamma \vdash !M : !A}\,(!I) \qquad \frac{\Gamma \vdash M : !B \quad \Gamma', x : B \vdash N : A}{\Gamma, \Gamma' \vdash let\ M\ be\ !x\ in\ N : A}\,(!E)$$

$$\frac{\Gamma \vdash M : !A \quad \Gamma', x : !A, y : !A \vdash N : C}{\Gamma, \Gamma' \vdash copy\ M\ as\ x @ y\ in\ N : C}\,(C) \qquad \frac{\Gamma \vdash M : !A \quad \Gamma' \vdash N : C}{\Gamma, \Gamma' \vdash kill\ M\ in\ N : C}\,(W)$$

Fig. 5. Typing schemes for λ^l

a left identity for instruction composition: $\epsilon;\mathcal{B}$ and $\mathcal{B}$ are identified in the metalanguge. A λ^l variable is just compiled to ϵ since when suffixed with the return instruction we shall obtain $\texttt{Return}(x)$. An integer constant is compiled into a register assignment instruction. The remaining cases are self explanatory. Notice that both $M\&N$ and $!M$ are compiled into code that generates appropriate closures. Also, registers appearing on the right-hand side that do not occur on the left-hand side are assumed to be fresh (for example, as in the third and fourth clauses).

By construction the following *type-preservation* result holds immediately.

Lemma 4.1 *If $\Gamma \vdash M : A$ is provable for some typing context Γ, preterm $M \in \mathcal{T}_U$ and type A, then the sequent $\Gamma \vdash \textsc{Comp}(U \mid M \mid z);\texttt{Return}(z) : A$ is provable in SS^c.*

We now address correctness of the LLAM. *Evaluation* in λ^l is defined by the standard call-by-value *evaluation schemes* (cf. Fig. 7). We say a *closed* λ^l-term M evaluates to a *canonical form* F if $M \Downarrow F$, where canonical forms are given by the grammar

$$F, G, H ::= \underline{n} \mid \lambda x : A.M \mid !M \mid M\&N \mid F \otimes G$$

The main obstacle towards a proof of correctness (cf. Thm 4.11) is that evaluation relies on substitution over λ^l-terms, whereas there is no such notion of substitution in the abstract machine. Moreover, in contrast to SECD-style

$$
\begin{array}{ll}
\textsc{Comp}(\{x\} \mid x \mid x) & \stackrel{\text{def}}{=} \epsilon \\
\textsc{Comp}(\emptyset \mid \underline{n} \mid z) & \stackrel{\text{def}}{=} z{=}\underline{n} \\
\textsc{Comp}(U \mid \lambda x : A.M \mid z) & \stackrel{\text{def}}{=} z{=}\texttt{makeClos}(\textsc{Comp}(U, x \mid M \mid y); \texttt{Return}(y), x, U) \\
\textsc{Comp}(U, V \mid M\, N \mid z) & \stackrel{\text{def}}{=} \textsc{Comp}(V \mid N \mid y_1); \textsc{Comp}(U \mid M \mid y_2); z{=}\texttt{call}\, y_2\, \texttt{with}\, y_1 \\
\textsc{Comp}(U, V \mid M \otimes N \mid z) & \stackrel{\text{def}}{=} \textsc{Comp}(U \mid M \mid y_1); \textsc{Comp}(V \mid N \mid y_2); z{=}\texttt{pair}(y_1, y_2) \\
\textsc{Comp}(U, V \mid \mathit{let}\, M\, \mathit{be}\, x \otimes y\, \mathit{in}\, N \mid z) & \stackrel{\text{def}}{=} \textsc{Comp}(U \mid M \mid w); (x,y){=}\texttt{unpair}(w); \textsc{Comp}(V, x, y \mid N \mid z) \\
\textsc{Comp}(U \mid M \& N \mid z) & \stackrel{\text{def}}{=} z{=}\texttt{makeClos}(\textsc{Comp}(U \mid M \mid y_1); \texttt{Return}(y_1), \\
 & \qquad \textsc{Comp}(U \mid N \mid y_2); \texttt{Return}(y_2), U) \\
\textsc{Comp}(U \mid \mathit{fst}(M) \mid z) & \stackrel{\text{def}}{=} \textsc{Comp}(U \mid M \mid y); z{=}\texttt{fst}(y) \\
\textsc{Comp}(U \mid \mathit{snd}(M) \mid z) & \stackrel{\text{def}}{=} \textsc{Comp}(U \mid M \mid y); z{=}\texttt{snd}(y) \\
\textsc{Comp}(U \mid !M \mid z) & \stackrel{\text{def}}{=} z{=}\texttt{makeOCClos}(\textsc{Comp}(U \mid M \mid y); \texttt{Return}(y), U) \\
\textsc{Comp}(U, V \mid \mathit{let}\, M\, \mathit{be}\, !x\, \mathit{in}\, N \mid z) & \stackrel{\text{def}}{=} \textsc{Comp}(U \mid M \mid y); x{=}\texttt{read}(y); \textsc{Comp}(V, x \mid N \mid z) \\
\textsc{Comp}(U, V \mid \mathit{copy}\, M\, \mathit{as}\, x @ y\, \mathit{in}\, N \mid z) & \stackrel{\text{def}}{=} \textsc{Comp}(U \mid M \mid w); (x,y){=}\texttt{copy}\, w; \textsc{Comp}(V, x, y \mid N \mid z) \\
\textsc{Comp}(U, V \mid \mathit{kill}\, M\, \mathit{in}\, N \mid z) & \stackrel{\text{def}}{=} \textsc{Comp}(U \mid M \mid w); \texttt{kill}(w); \textsc{Comp}(V \mid N \mid z)
\end{array}
$$

Fig. 6. Compilation function

$$
\frac{}{\lambda x : A.M \Downarrow \lambda x : A.M} \qquad \frac{N \Downarrow F \quad M \Downarrow \lambda x : A.P \quad P\{x \leftarrow F\} \Downarrow G}{M\, N \Downarrow G}
$$

$$
\frac{M \Downarrow F \quad N \Downarrow G}{M \otimes N \Downarrow F \otimes G} \qquad \frac{M \Downarrow F \otimes G \quad N\{x, y \leftarrow F, G\} \Downarrow H}{\mathit{let}\, M\, \mathit{be}\, x \otimes y\, \mathit{in}\, N \Downarrow H}
$$

$$
\frac{}{M \& N \Downarrow M \& N} \qquad \frac{M \Downarrow P \& Q \quad P \Downarrow F}{\mathit{fst}(M) \Downarrow F} \qquad \frac{M \Downarrow P \& Q \quad Q \Downarrow F}{\mathit{snd}(M) \Downarrow F}
$$

$$
\frac{}{!M \Downarrow !M} \qquad \frac{M \Downarrow !P \quad P \Downarrow F \quad N\{x \leftarrow F\} \Downarrow G}{\mathit{let}\, M\, \mathit{be}\, !x\, \mathit{in}\, N \Downarrow G}
$$

$$
\frac{M \Downarrow !P \quad N\{x, y \leftarrow !P, !P\} \Downarrow F}{\mathit{copy}\, M\, \mathit{as}\, x @ y\, \mathit{in}\, N \Downarrow F} \qquad \frac{M \Downarrow !P \quad N \Downarrow F}{\mathit{kill}\, M\, \mathit{in}\, N \Downarrow F} \qquad \frac{}{\underline{n} \Downarrow \underline{n}}
$$

Fig. 7. Evaluation for λ^l

abstract machines, the LLAM executes low-level code and this adds further complications. Indeed, in SECD-style machines the code executed by the machine is a lambda expression itself and this makes the task of establishing correctness results easier. Thus we define an intermediate (big-step) operational semantics, called *lifted evaluation*, in which the use of substitution is replaced in favor of an *assignment* with which evaluation schemes are decorated. In turn this requires adapting the notions of canonical form. Essentially, we obtain a weak linear calculus with explicit substitutions.

$$\frac{}{(\lambda x : A.M)[\sigma] \Downarrow (\lambda x : A.M)[\sigma]} \qquad \frac{N[\sigma_N] \Downarrow C \quad M[\sigma_M] \Downarrow (\lambda x : A.P)[\tau] \quad P[\tau \oplus \{x := C\}] \Downarrow D}{(M\,N)[\sigma] \Downarrow D}$$

$$\frac{M[\sigma_M] \Downarrow C \quad N[\sigma_N] \Downarrow D}{(M \otimes N)[\sigma] \Downarrow C \otimes D} \qquad \frac{M[\sigma_M] \Downarrow C \otimes D \quad N[\sigma_N \oplus \{x := C, y := D\}] \Downarrow E}{(let\ M\ be\ x \otimes y\ in\ N)[\sigma] \Downarrow E}$$

$$\frac{}{(M\&N)[\sigma] \Downarrow (M\&N)[\sigma]} \qquad \frac{M[\sigma] \Downarrow (P\&Q)[\tau] \quad P[\tau] \Downarrow C}{fst(M)[\sigma] \Downarrow C} \qquad \frac{M[\sigma] \Downarrow (P\&Q)[\tau] \quad Q[\tau] \Downarrow C}{snd(M)[\sigma] \Downarrow C}$$

$$\frac{}{(!M)[\sigma] \Downarrow (!M)[\sigma]} \qquad \frac{M[\sigma_M] \Downarrow (!P)[\tau] \quad P[\tau] \Downarrow C \quad N[\sigma_N \oplus \{x := C\}] \Downarrow D}{(let\ M\ be\ !x\ in\ N)[\sigma] \Downarrow D}$$

$$\frac{M[\sigma_M] \Downarrow (!P)[\tau] \quad N[\sigma_N \oplus \{x := (!P)[\tau], y := (!P)[\tau]\}] \Downarrow C}{(copy\ M\ as\ x@y\ in\ N)[\sigma] \Downarrow C} \qquad \frac{M[\sigma_M] \Downarrow (!P)[\tau] \quad N[\sigma_N] \Downarrow C}{(kill\ M\ in\ N)[\sigma] \Downarrow C}$$

$$\frac{}{x[\{x := C\}] \Downarrow C} \qquad \frac{}{\underline{n}[\emptyset] \Downarrow \underline{n}}$$

Fig. 8. Lifted Evaluation for λ^l

Assignments and *lifted canonical forms* are defined as follows

$$\sigma, \tau ::= \{x_1 := C_1, \ldots, x_n := C_n\}$$
$$C, D ::= \underline{n} \mid (\lambda x : A.M)[\sigma] \mid (!M)[\sigma] \mid (M\&N)[\sigma] \mid C \otimes D$$

We write $\mathsf{Dom}(\sigma)$ for the domain of σ. We write $\sigma \oplus \{x := C\}$ for the extension of σ with $x := C$ under the assumption that $x \notin \mathsf{Dom}(\sigma)$. If M is a λ^l-term and $\mathsf{FV}(M) \subseteq \mathsf{Dom}(\sigma)$, then σ_M is the restriction of σ to the free variables of M. *Lifted evaluation* of λ^l-terms is defined by the *evaluation schemes* of Fig. 8. Given a preterm $M \in \mathcal{T}_V$ and an assignment σ such that $\mathsf{Dom}(\sigma) = V$, if the relation $(M)[\sigma] \Downarrow C$ holds, then we say that M *l-evaluates* to C *under* σ. The expression $M[\sigma]$ may be interpreted as term M with pending assignment σ. In contrast to evaluation, lifted evaluation does not rely on applying substitution but rather records the substitution until it is required. This is witnessed, for example, by comparing the evaluation scheme for application with the corresponding lifted one. This is also reflected in canonical forms: lifted canonical forms include, for all canonical forms corresponding to lazy types and to the function type, a suspended substitution. All in all, the resulting evaluation mechanism is closer to the abstract machine.

The proof of correctness proceeds in three steps. First we establish the correspondence between evaluation and lifted evaluation (Prop. 4.3). Then we prove a correctness result for lifted evaluation (Prop. 4.7). Finally, appealing to the result of the first step we prove correctness for evaluation (Thm 4.11).

4.1 Relating Evaluation and Lifted Evaluation

We begin by relating canonical forms and lifted canonical forms. The idea is that C is a lifting of F if F results from "flattening" C by applying all pending assignments.

Definition 4.2 *We say C is a lifting of F if $\overline{C} = F$, where*

$$\overline{\underline{n}} \stackrel{\text{def}}{=} \underline{n} \qquad \overline{(\lambda x : A.M)[\sigma]} \stackrel{\text{def}}{=} \lambda x : A.M_{\overline{\sigma \oplus \{x:=x\}}} \qquad \overline{(!M)[\sigma]} \stackrel{\text{def}}{=} !M_{\overline{\sigma}}$$

$$\overline{(M\&N)[\sigma]} \stackrel{\text{def}}{=} M_{\overline{\sigma}}\&N_{\overline{\sigma}} \qquad \overline{C \otimes D} \stackrel{\text{def}}{=} \overline{C} \otimes \overline{D}$$

The notation $M_{\overline{\sigma}}$ (defined simultaneously with $\overline{C}$) denotes the term resulting from M by applying (the flattening of) σ and has the following defining clauses:

$$x_{\overline{\{x:=C\}}} \stackrel{\text{def}}{=} \overline{C} \qquad x_{\overline{\{x:=x\}}} \stackrel{\text{def}}{=} x \qquad \underline{n}_{\overline{\emptyset}} \stackrel{\text{def}}{=} \underline{n}$$

$$(\lambda x : A.M)_{\overline{\sigma}} \stackrel{\text{def}}{=} \lambda x : A.M_{\overline{\sigma \oplus \{x:=x\}}} \qquad (M\,N)_{\overline{\sigma}} \stackrel{\text{def}}{=} M_{\overline{\sigma_M}}\,N_{\overline{\sigma_N}}$$

$$(M \otimes N)_{\overline{\sigma}} \stackrel{\text{def}}{=} M_{\overline{\sigma_M}} \otimes N_{\overline{\sigma_N}} \qquad fst(M)_{\overline{\sigma}} \stackrel{\text{def}}{=} fst(M_{\overline{\sigma}}) \qquad snd(M)_{\overline{\sigma}} \stackrel{\text{def}}{=} snd(M_{\overline{\sigma}})$$

$$(M\&N)_{\overline{\sigma}} \stackrel{\text{def}}{=} M_{\overline{\sigma}}\&N_{\overline{\sigma}} \qquad (!M)_{\overline{\sigma}} \stackrel{\text{def}}{=} !M_{\overline{\sigma}}$$

$$(let\ M\ be\ !x\ in\ N)_{\overline{\sigma}} \stackrel{\text{def}}{=} let\ M_{\overline{\sigma_M}}\ be\ !x\ in\ N_{\overline{\sigma_N \oplus \{x:=x\}}}$$

$$(copy\ M\ as\ x@y\ in\ N)_{\overline{\sigma}} \stackrel{\text{def}}{=} copy\ M_{\overline{\sigma_M}}\ as\ x@y\ in\ N_{\overline{\sigma_N \oplus \{x,y:=x,y\}}}$$

$$(kill\ M\ in\ N)_{\overline{\sigma}} \stackrel{\text{def}}{=} kill\ M_{\overline{\sigma_M}}\ in\ N_{\overline{\sigma_N}}$$

$$(let\ M\ be\ x \otimes y\ in\ N)_{\overline{\sigma}} \stackrel{\text{def}}{=} let\ M_{\overline{\sigma_M}}\ be\ x \otimes y\ in\ N_{\overline{\sigma_N \oplus \{x,y:=x,y\}}}$$

Here σ is, in fact, an *extended assignment* in that components of the form $x_i := x_i$ are admitted.

Lifted evaluation preserves evaluation. Its proof relies on the fact that if C is a lifting of a closed canonical form F, then $P_{\overline{\sigma \oplus \{x:=x\}}}\{x \leftarrow F\} = P_{\overline{\sigma \oplus \{x:=C\}}}$; it proceeds by induction on $M \Downarrow F$.

Proposition 4.3 *$M \Downarrow F$, for M a closed term, implies for all assignments σ and preterms O such that $\mathsf{FV}(O) = \mathsf{Dom}(\sigma)$ and $O_{\overline{\sigma}} = M$ there exists a lifted canonical form C such that $\overline{C} = F$ and $O[\sigma] \Downarrow C$.*

Taking O to be M and σ to be $\emptyset$ (the identity substitution) in Prop. 4.3 we deduce:

Corollary 4.4 *If $M \Downarrow F$, for M a closed term, then there exists a lifting C of F such that $M[\emptyset] \Downarrow C$.*

Remark 4.5 *The converse also holds (if $M[\sigma] \Downarrow C$, then $M_{\overline{\sigma}} \Downarrow \overline{C}$) however we do not make use of it in this work.*

4.2 Correctness for Lifted Evaluation

Before stating the main result of this section we introduce a definition.

$$
\begin{aligned}
\underline{n}^{\mathsf{Comp}} &\stackrel{\text{def}}{=} \underline{n} \\
((\lambda x : A.M)[\sigma])^{\mathsf{Comp}} &\stackrel{\text{def}}{=} Clos(\textsc{Comp}(\mathsf{Dom}(\sigma), x \mid M \mid z);\mathtt{Return}(z), x, \sigma^{\mathsf{Comp}}) \\
(!M)[\sigma]\}^{\mathsf{Comp}} &\stackrel{\text{def}}{=} OCClos(\textsc{Comp}(\mathsf{Dom}(\sigma) \mid M \mid z);\mathtt{Return}(z), \sigma^{\mathsf{Comp}}) \\
(M\&N)[\sigma])^{\mathsf{Comp}} &\stackrel{\text{def}}{=} LPCLos(\textsc{Comp}(\mathsf{Dom}(\sigma) \mid M \mid z);\mathtt{Return}(z), \\
&\qquad \textsc{Comp}(\mathsf{Dom}(\sigma) \mid N \mid z);\mathtt{Return}(z), \sigma^{\mathsf{Comp}}) \\
(C \otimes D)^{\mathsf{Comp}} &\stackrel{\text{def}}{=} \langle C^{\mathsf{Comp}}, D^{\mathsf{Comp}} \rangle \\
\{x_1 := C_1, \ldots, x_n := C_n\}^{\mathsf{Comp}} &\stackrel{\text{def}}{=} \{x_1 := C_1^{\mathsf{Comp}}, \ldots, x_n := C_n^{\mathsf{Comp}}\}
\end{aligned}
$$

Fig. 9. Compilation of canonical forms

Definition 4.6 *The compilation of a lifted canonical form C, denoted C^{Comp}, is the value (cf. Sec. 3.2) defined by the clauses given in Fig. 9*[5] *:*

The main result of this section reads as follows:

Proposition 4.7 *If $M[\sigma] \Downarrow C$, then for any fresh register z and code block variable X, $\langle \textsc{Comp}(\mathsf{Dom}(\sigma) \mid M \mid z);X, \sigma^{\mathsf{Comp}}, D\rangle \twoheadrightarrow \langle X, \{z := C^{\mathsf{Comp}}\}, D\rangle$.*

The rest of this section introduces the notions and results required to prove Prop. 4.7. We begin by introducing partial machine states and then present the Instantiation Lemma, crucial to the proof of Prop. 4.7.

Let X, Y, Z stand for code block variables. An *open code block* is one that has the form $\iota_1; \ldots; \iota_n; X$ (by abuse of notation we write $\mathcal{B};X$ and let $\mathcal{B}$ stand for $\iota_1; \ldots; \iota_n$).

Definition 4.8 (Partial state) *A partial state (for (X, D)) is a machine state S of one of two forms:*

- *$\langle \mathcal{B};X, R, D\rangle$, for some code block $\mathcal{B}$ and register bank R, or*
- *$\langle \mathcal{B}_1, R_1, [\mathcal{B}_2, R_2, x_2] \cdot \ldots \cdot [\mathcal{B}_n, R_n, x_n] \cdot [\mathcal{B};X, R, x] \cdot D\rangle$ for some code blocks $\mathcal{B}, \mathcal{B}_i$ $(1 \leq i \leq n)$, register banks R, R_i $(1 \leq i \leq n)$ and registers x, x_i $(2 \leq i \leq n)$.*

where the sole occurrence of X is the indicated one (other open code blocks may occur in D though). In both cases we say R is the register bank of S.

Computation commencing at a partial state for (X, D) is seen never to access or modify D and, if well-typed[6], proceeds until it blocks at X. Indeed, one may verify the following by case analysis on the reduction schemes of the LLAM.

Lemma 4.9 *If S is a partial state for (X, D) and $S \rightarrow S'$, then S' is a partial*

[5] Strictly speaking, the compilation of a lifted canonical form is parameterized over some variable z. We pick such a variable according to the context of application.

[6] In which case the occurrences of code block variables, representing an "incomplete" proof in SS, are given appropriate types.

state for (X, D) *(ie. for the same* X *and* D*).*

Given a partial state S we may instantiate the code block variable with a proper code block. The following *instantiation function* does just that and, moreover, extends the register bank of S with additional mappings. Below we write $R \oplus R'$ for the union of the register banks R, R' which we assume to have disjoint domains. Let $\mathcal{D}_i$ stand for a suspended procedure activation $[\mathcal{B}_i, R_i, x_i]$, $i \in 2..n$.

$$\mathsf{I}_{\mathcal{B}',R'}(\langle \mathcal{B}; X, R, D\rangle) \stackrel{\text{def}}{=} \langle \mathcal{B};\mathcal{B}', R \oplus R', D\rangle$$
$$\mathsf{I}_{\mathcal{B}',R'}(\langle \mathcal{B}_1, R_1, \mathcal{D}_2 \cdot \ldots \cdot \mathcal{D}_n \cdot [\mathcal{B}; X, R, x] \cdot D\rangle) \stackrel{\text{def}}{=} \langle \mathcal{B}_1, R_1, \mathcal{D}_2 \cdot \ldots \cdot \mathcal{D}_n \cdot [\mathcal{B};\mathcal{B}', R \oplus R', x] \cdot D\rangle$$

Let $\mathsf{Reg}(\mathcal{B})$ denote the set of registers occurring in code block $\mathcal{B}$. We say a register bank R' is *compatible* with S, a partial machine state for (X, D), if the following holds:

- If $S = \langle \mathcal{B}; X, R, D\rangle$, then $\mathsf{Reg}(\mathcal{B}) \cap \mathsf{Dom}(R') = \emptyset$ and $\mathsf{Dom}(R) \cap \mathsf{Dom}(R') = \emptyset$.
- If $S = \langle \mathcal{B}_1, R_1, [\mathcal{B}_2, R_2, x_2] \cdot \ldots \cdot [\mathcal{B}_n, R_n, x_n] \cdot [\mathcal{B}; X, R, x] \cdot D\rangle$, then $(\mathsf{Reg}(\mathcal{B}) \cup \{x\}) \cap \mathsf{Dom}(R') = \emptyset$ and $\mathsf{Dom}(R) \cap \mathsf{Dom}(R') = \emptyset$.

Lemma 4.10 (Instantiation) *If* S *is a partial state,* R' *is compatible with* S *and* $S \rightarrowtail S'$*, then* $\mathsf{I}_{\mathcal{B}',R'}(S) \rightarrowtail \mathsf{I}_{\mathcal{B}',R'}(S')$.

Proof. By case analysis for one-step reduction using the fact that compatibility is preserved by reduction and then extended by induction to many-step reduction.

We now address the proof of Prop. 4.7. It proceeds by induction on the proof of $M[\sigma] \Downarrow C$. Here we consider a sample case.

Proof. Let us call S the machine state $\langle \textsc{Comp}(\mathsf{Dom}(\sigma) \mid M \mid z); X, \sigma^{\mathsf{Comp}}, D\rangle$. Suppose the proof ends in:

$$\frac{N[\sigma_N] \Downarrow C \quad M[\sigma_M] \Downarrow (\lambda x : A.P)[\tau] \quad P[\tau \oplus \{x := C\}] \Downarrow E}{(M\,N)[\sigma] \Downarrow E}$$

The idea of the proof is to apply the IH on each of the hypothesis. This yields isolated reductions in the LLAM for the compilation of N under register bank σ_N^{Comp}, M under register bank σ_M^{Comp} and P under register bank $\tau^{\mathsf{Comp}} \oplus \{x := C^{\mathsf{Comp}}\}$, respectively. The Instantiation Lemma is then resorted to in order to weave these reductions into a unique reduction of the compilation of $M\,N$ under register bank σ^{Comp}.

Let U stand for $\mathsf{Dom}(\sigma_M)$ and V for $\mathsf{Dom}(\sigma_N)$. S takes the form $\langle \textsc{Comp}(V \mid N \mid y_1); \textsc{Comp}(U \mid M \mid y_2); z\texttt{=call}\, y_2\, \texttt{with}\, y_1; X, \sigma^{\mathsf{Comp}}, D\rangle$. By the IH applied to $N[\sigma_N] \Downarrow C$ we deduce $\langle \textsc{Comp}(V \mid N \mid y_1); Y, \sigma_N^{\mathsf{Comp}}, D\rangle \rightarrowtail \langle Y, \{y_1 := C^{\mathsf{Comp}}\}, D\rangle$. Therefore, by the Instantiation Lemma ($\mathcal{B}' = \textsc{Comp}(U \mid M \mid y_2); z\texttt{=call}\, y_2\, \texttt{with}\, y_1; X$ and $R' = \sigma_M$) and noting that $\sigma^{\mathsf{Comp}} = \sigma_M^{\mathsf{Comp}} \oplus \sigma_N^{\mathsf{Comp}}$

$$S \rightarrow \langle \textsc{Comp}(U \mid M \mid y_2); z{=}\texttt{call}\ y_2\ \texttt{with}\ y_1\,; X, \sigma_M \oplus \{y_1 := C^{\mathsf{Comp}}\}, D\rangle \tag{2}$$

By the IH applied to $M[\sigma_M] \Downarrow (\lambda x : A.P)[\tau]$ we have

$$\langle \textsc{Comp}(U \mid M \mid y_2); Z, \sigma_M^{\mathsf{Comp}}, D\rangle \rightarrow \langle Z, \{y_2 := ((\lambda x : A.P)[\tau])^{\mathsf{Comp}}\}, D\rangle =$$
$$= \langle Z, \{y_2 := \mathit{Clos}(\textsc{Comp}(\mathsf{Dom}(\sigma), x \mid P \mid w); \texttt{Return}(w), x, \tau^{\mathsf{Comp}})\}, D\rangle$$

By the Instantiation Lemma ($\mathcal{B}' = z{=}\texttt{call}\ y_2\ \texttt{with}\ y_1\,; X$ and $R' = \{y_1 := C^{\mathsf{Comp}}\}$) and a further reduction step yields

$$\begin{aligned} &\langle \textsc{Comp}(U \mid M \mid y_2); z{=}\texttt{call}\ y_2\ \texttt{with}\ y_1\,; X, \sigma_M^{\mathsf{Comp}} \oplus \{y_1 := C^{\mathsf{Comp}}\}, D\rangle \\ \rightarrow\ &\langle z{=}\texttt{call}\ y_2\ \texttt{with}\ y_1\,; X, \rho, D\rangle \\ \rightarrow\ &\langle \textsc{Comp}(\mathsf{Dom}(\sigma), x \mid P \mid w); \texttt{Return}(w), \tau^{\mathsf{Comp}}[x := C^{\mathsf{Comp}}], [X, \emptyset, z] \cdot D\rangle \end{aligned} \tag{3}$$

where ρ is the register bank $\{y_1 := C^{\mathsf{Comp}}, y_2 := \mathit{Clos}(\textsc{Comp}(\mathsf{Dom}(\sigma), x \mid P \mid w); \texttt{Return}(w), x, \tau^{\mathsf{Comp}})\}$.

A new application of the IH, this time to $P[\tau \oplus \{x := C\}] \Downarrow E$, yields

$$\langle \textsc{Comp}(\mathsf{Dom}(\tau), x \mid P \mid w); W, \tau^{\mathsf{Comp}}[x := C^{\mathsf{Comp}}], [X, \emptyset, z] \cdot D\rangle \rightarrow \langle W, \{w := E^{\mathsf{Comp}}\}, [X, \emptyset, z] \cdot D\rangle$$

Again we resort to the Instantiation Lemma ($\mathcal{B}' = \texttt{Return}(w)$ and $R' = \emptyset$) and an additional reduction step in order to obtain

$$\begin{aligned} &\langle \textsc{Comp}(\mathsf{Dom}(\tau), x \mid P \mid w); \texttt{Return}(w), \tau^{\mathsf{Comp}}[x := C^{\mathsf{Comp}}], [X, \emptyset, z] \cdot D\rangle \\ \rightarrow\ &\langle \texttt{Return}(w), \{w := E^{\mathsf{Comp}}\}, [X, \emptyset, z] \cdot D\rangle \rightarrow \langle X, \{z := E^{\mathsf{Comp}}\}, D\rangle \end{aligned} \tag{4}$$

We conclude by juxtaposing reduction sequences (2), (3) and (4).

We now address the main result.

Theorem 4.11 (Correctness) *If $M \Downarrow F$, then there exists a lifting C of F such that $\langle \textsc{Comp}(\emptyset \mid M \mid z); \texttt{Return}(z), \emptyset, \emptyset\rangle \rightarrow \langle \epsilon, \{z := C^{\mathsf{Comp}}\}, \emptyset\rangle$.*

Proof. If $M \Downarrow F$ with M closed λ^l-term, then by Corollary 4.4 there exists a lifted canonical form C such that $M[\emptyset] \Downarrow C$ and $\overline{C} = F$. By Prop. 4.7, for any fresh register z and fresh code block variable X $\langle \textsc{Comp}(\emptyset \mid M \mid z); X, \emptyset, \emptyset\rangle \rightarrow \langle X, \{z := C^{\mathsf{Comp}}\}, \emptyset\rangle$. Finally, by the Instantiation Lemma and an additional reduction step $\langle \textsc{Comp}(\emptyset \mid M \mid z); \texttt{Return}(z), \emptyset, \emptyset\rangle \rightarrow \langle \texttt{Return}(z), \{z := C^{\mathsf{Comp}}\}, \emptyset\rangle \rightarrow \langle \epsilon, \{z := C^{\mathsf{Comp}}\}, \emptyset\rangle$.

5 Conclusions

We have presented an abstract machine based on ILL that executes low-level code. This machine has been derived from cut-elimination of a Sequent Calculus presentation (SS) of ILL. It is a register based abstract machine (a stack based machine is easily obtainable by treating contexts as sequences rather than multisets) whose states consist of a low-level code block, a register bank and a dump containing suspended procedure activations. Translation of Natural Deduction proofs to SS yields a type-preserving compilation function from

the Linear Lambda Calculus [24] (λ^l) to low-level code. We prove that the LLAM is correct with respect to the standard call-by-value natural semantics (evaluation) of λ^l.

An issue that warrants a further look is de-compilation of low-level code to terms in λ^l. This should be possible in the same way that it has been addressed for bytecode and low-level code based on Intuitionistic Propositional Logic [11]. Also, we have not addressed sharing in our abstract machine. Indeed, the LLAM trivially satisfies the single-pointer property since all closures are recomputed. Although our efforts have concentrated on providing a robust logical foundation for an abstract machine based on linear logic, we are aware that sharing is an important topic which must be addressed and leave further investigation on this matter to future work. On a related note, Danvy [6] has developed a general technique to "mechanically deconstruct" the SECD machine into an evaluator and to construct a SECD-like abstract machine from an evaluator. He makes use of well-known implementation techniques: defunctionalization, CPS conversion and closure conversion. It would be interesting to see if these phases could be recast in proof theoretical terms.

Finally, the there is a striking similarity between the sequential sequent calculus (SS) and Shroeder-Heister's presentation [21] of intuitionist propositional logic based on higher-order natural deduction (in addition to the standard notion of assumptions there are assumption *rules* that may also be discharged). The natural deduction rule for implication elimination takes the form

$$\dfrac{A \supset B \qquad \begin{array}{c} A \\ \hline B \\ \hline C \end{array}}{C}\,(\supset E)$$

Here the assumption rule $\dfrac{A}{B}$ may be used in the upper right hand proof but is discharged by $(\supset E)$. The reader is invited to compare $(\supset E)$ with $(\multimap R)$. An in-depth analysis is postponed to future work.

Acknowledgement

To Peter Thiemann for pointing out Danvy's article [6] to me and to the anonymous referees for helping improve the paper.

References

[1] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A functional correspondence between evaluators and abstract machines. In Dale Miller, editor, *Proc. of the Fifth ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 8–19. ACM Press, 2003.

[2] Samson Abramsky. Computational interpretations of linear logic. *TCS*, 111:3–57, 1993.

[3] Zena Ariola, Aaron Bohannon, and Amr Sabry. Sequent calculi and abstract machines. Submitted for publication.

[4] Francisco Alberti and Eike Ritter. An efficient linear abstract machine with single-pointer property. Presented at ESSLLI'98 Workshop on Logical Abstract Machines, 1998.

[5] Nick Benton, Gavin Bierman, Valeria de Paiva, and Martin Hyland. A term calculus for intuitionistic linear logic. In *Proc. of Typed Lambda Calculi and Applications*, LNCS. Springer-Verlag, 1993.

[6] Olivier Danvy. A rational deconstruction of Landin's SECD machine. In *Proceedings of the International Workshop on the Implementation and Application of Functional Languages 2004*, volume 3474 of *LNCS*, pages 52–71, 2004.

[7] Vincent Danos, Hugo Herbelin, and Laurent Regnier. Games semantics and abstract machines. In *Proc. of LICS'96*, 1996.

[8] Stephen Diehl, Pieter Hartel, and Peter Sestoft. Abstract machines for programming language implementation. *Future Generation Computer Systems*, 16(7):739–751, 2000.

[9] Jean Gallier. On Girard's "Candidats de Reductibilité". In Odifreddi, editor, *Logic and Computer Science*, pages 123–203. Academic Press, 1990.

[10] Jean-Yves Girard. Linear logic. *TCS*, 50(1), 1987.

[11] Shinya Katsumata and Atsushi Ohori. Proof-directed de-compilation of low-level code. In *Proceedings of European Sympoisum on Programming*, volume 2028 of *LNCS*. Springer-Verlag, 2001.

[12] Yves Lafont. The linear abstract machine. *TCS*, 59:157–180, 1988.

[13] Peter Landin. The mechanical evaluation of expressions. *Computer Journal*, 6:308–320, 1964.

[14] Patrick Lincoln and John Mitchell. Operational aspects of the linear lambda calculus. In *Proc. of the IEEE Symposium on Logic in Computer Science*. IEEE Press, 1992.

[15] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, second edition edition, 1999.

[16] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to Typed Assembly Language. *ACM Transactions on Programming Languages and Systems*, 21(3):528–569, May 1999.

[17] Atsushi Ohori. The logical abstract machine: A curry-howard isomorphism for machine code. In *Proceedings of the Fuji International Symposium on Functional and Logic Programming*, volume 1722, pages 300–318, 1999.

[18] Atsushi Ohori. A proof theory for machine code. Submitted for publication, 2005.

[19] Gordon Plotkin. Call-by-name, call-by value and the lambda calculus. *TCS*, 1:125–159, 1975.

[20] Christophe Raffalli. Machine deduction. In *Proceedings of Types for Proofs and Programs*, volume 806 of *LNCS*, pages 333–351, 1994.

[21] Peter Schroeder-Heister. A natural extension of natural deduction. *Journal of Symbolic Logic*, 49:1284–1300, 1984.

[22] David N. Turner and Philip Wadler. Operational interpretations of linear logic. TCS, to appear.

[23] Philip Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *IFIP Working Conference on Programming Concepts and Methods*, April 1990.

[24] Philip Wadler. There's no substitute for linear logic. In *Proceedings 8'th International Workshop on the Mathematical Foundations of Programming Semantics*, 1992.

[25] David Wakeling and Colin Runciman. Linearity and laziness. In *Proc. of the 5th ACM conference on Functional Programming Languages and Computer Architecture*, pages 215–240, 1991.